NON-BLOCKING LOAD BUFFER AND A MULTIPLE-PRIORITY MEMORY SYSTEM FOR REAL-TIME MULTIPROCESSING

FIELD OF THE INVENTION

The present invention relates to interfacing between a microprocessor,

5   peripherals and/or memories. More particularly, the present invention relates to a

data processing system for interfacing a high-speed data bus, which is connected to

one or more processors and a cache, to a second, possibly lower speed peripheral

bus, which is connected to a plurality of peripherals and memories, by a non-

blocking load buffer so that a continuous maximum throughput from multiple I/O

10   devices is provided via programmed I/O (memory-mapped I/O) of the processors

and a plurality of non-blocking loads may be simultaneously performed. Also, the

present invention is directed to a multiple-priority non-blocking load buffer which

serves a multiprocessor for running real-time processes of varying deadlines by

priority-based non-FIFO scheduling of memory and peripheral accesses.

BACKGROUND OF THE INVENTION

Conventionally, microprocessor and memory system applications retrieve

data from relatively low bandwidth I/O devices, process the data by a processor and

then store the processed data to the low bandwidth I/O devices.   In typical

microprocessor memory system applications, a processor and a cache are directly

20   coupled by a bus to a plurality of peripherals such as I/O devices and memory

devices.   However, the processor may be unnecessarily idled due to the latency

between the I/O devices and the processor, thus causing the processor to stall and,

as a result, excessive time is required to complete tasks.

In known processing systems, when an operation is performed on one of the

25   peripherals such as a memory device, the time between performing such an

operation and subsequent operations is dependent upon the latency period of the

memory device. Thereby, the processor will be stalled during the entire duration of

the memory transaction. One solution for improving the processing speed of known

processing systems is to perform additional operations during the time of the latency period as long as there is no load-use dependency upon the additional operations. For example, if data is loaded into a first register by a first operation (1), where the first operation (1) corresponds to:

$$\text{load } r1 \leftarrow [r2] \quad (1)$$

and the first register is added to another operation by a second operation (2) where the second operation (2) corresponds to:

$$\text{add } r3 \leftarrow r1 + r4 \quad (2)$$

the operation (2) is load-use dependent and the operation (2) must wait for the latency period before being performed. Figures 1(a) and 1(b) illustrate a load-use dependent operation where the time for initiating the operation (2) must wait until the operation (1) is completed. Operation (2) is dependent upon the short latency period t_1 in Figure 1(a) corresponding to a fast memory and the long latency period t_2 in Figure 1(b) corresponds to a slower memory.

A non-blocking cache and a non-blocking processor are known where a load operation is performed and additional operations other than loads may be subsequently performed during the latency period as long as the operation is not dependent upon the initial load operation. Figures 2(a) and 2(b) illustrate such operations. In operation (1), the first register is loaded. Next, operations (1.1) and (1.2) are to be executed. As long as operations (1.1) and (1.2) are not load dependent on another load, these operations may be performed during the latency period t_2 as illustrated in Figure 2(a). However, if operation (1.1) is either load-dependent or a pending load, operations (1.1) and (1.2) must wait until the latency period t_2 ends before being performed.

Also known is a Stall-On-Use (Hit Under Miss) operation for achieving cache miss optimizations as described in "A 200 MFLOP Precision Architecture Processor" at Hot Chips IV, 1993, William Jaffe et al. In this Hit Under Miss operation, when one miss is outstanding only certain other types of instructions may be executed before the system stalls. For example, during the handling of a load

miss, execution proceeds until the target register is needed as an operand for another instruction or until another load miss occurs. However, this system is not capable of handling two misses being outstanding at the same time. For a store miss, execution proceeds until a load or sub-word store occurs to the missing line.

5   This Hit Under Miss feature can improve the runtime performance of general-purpose computing applications. Examples of programs that benefit from the Hit Under Miss feature are SPEC benchmarks, SPICE circuit simulators and gcc C compilers. However, the Hit Under Miss feature does not sufficiently meet the high I/O bandwidth requirements for digital signal processing applications such as

10   digital video, audio and RF processing.

Known microprocessor and memory system applications use real-time processes which are programs having deadlines corresponding to times where data processing must be completed. For example, an audio waveform device driver process must supply audio samples at regular intervals to the output buffers of the

15   audio device. When the driver software is late in delivering data for an audio waveform device driver, the generated audio may be interrupted by objectionable noises due to an output buffer underflowing.

In order to analyze whether or not a real-time process can meet its deadlines under all conditions requires predictability of the worst-case performance of the real-

20   time processing program. However, the sensitivity of the real-time processing program to its input data or its environment makes it impractical in many cases to exhaustively check the behavior of the process under all conditions. Therefore, the programmer must rely on some combination of analysis and empirical tests to verify that the real-time process will complete in the requisite time. The goals of real-time

25   processing tend to be incompatible with computing platforms that have memory or peripheral systems in which the latency of the transactions is unpredictable because an analysis of whether the real-time deadlines can be met may not be possible or worst-case assumptions of memory performance are required. For example, performance estimates can be made by assuming that every memory transaction

takes the maximum possible time.   However, such an assumption may be so pessimistic that any useful estimate for the upper bound on the execution time of a real-time task cannot be made. Furthermore, even if the estimates are only slightly pessimistic, overly conservative decisions will be made for the hardware

5   performance requirements so that a system results that is more expensive than necessary.

Also, it is especially difficult to reliably predict real-time processing performance on known multiprocessors because the memory and peripherals are not typically multi-ported. Therefore simultaneous access by two or more processors

10   to the same memory device must be serialized.   Even if a device is capable of handling multiple transactions in parallel, the bus shared by all of the processors may still serialize the transactions to some degree.

If memory requests are handled in a FIFO manner by a known multiprocessor, a memory transaction which arrives slightly later than another

15   memory transaction may take a much longer amount of time to complete since the later arriving memory requests must wait until the earlier memory request is serviced. Due to this sensitivity, very small changes in the memory access patterns of a program can cause large changes in its performance. This situation grows worse as more processors share the same memory. For example, if ten processors

20   attempt to access the same remote memory location simultaneously, the spread in memory latency among the processors might be 10:1 because as many as nine of these memory transactions may be buffered for later handling. In general, it is not possible to predict which of these processors will suffer the higher latencies and which of these processors will receive fast replies to their memory accesses. Very

25   small changes to a program or its input data may cause the program to exhibit slight operation differences which perturb the timing of the memory transactions.

Furthermore, types of memory which exhibit locality effects may exacerbate the above-described situation. For example, accesses to DRAMs are approximately two times faster if executed in page mode. To use page mode, a recent access must

have been made to an address in the same memory segment (page). One of the most common access patterns is sequential accesses to consecutive locations in memory. These memory patterns tend to achieve high page locality, thus achieving high throughput and low latency. Known programs which attempt to take advantage

5   of the benefits of page mode may be thwarted when a second program executing on another processor is allowed to interpose memory transactions on a different memory page. For instance, if ten processors, each with its own sequential memory access pattern, attempt to access the same DRAM bank simultaneously and each of the accesses is to a different memory page, the spread and memory latencies

10   between the fastest and slowest responses might be more than 25:1.

The present invention is directed to allowing a high rate of transfer to memory and I/O devices for tasks which have real-time requirements. The present invention is also directed to allowing the system to buffer I/O requests from several processors within a multiprocessor at once with a non-blocking load buffer.

15   Furthermore, the present invention is directed to extending the basic non-blocking load buffer to service a data processing system running real-time processes of varying deadlines by using scheduling of memory and peripheral accesses which is not strictly FIFO scheduling.

SUMMARY OF THE INVENTION

20   An object of the present invention is to reduce the effect of memory latency by overlapping a plurality of non-blocking loads during the latency period and to achieve a similar reduction on the effect of latency for non-blocking stores without requiring the additional temporary storage memory of a separate store buffer.

Another object of the present invention is to provide a non-blocking load

25   buffer which buffers I/O requests from one or more processors within a multiprocessor so that a plurality of I/O memory transactions, such as loads and stores, may be simultaneously performed and the high I/O bandwidth requirements for digital signal processing applications may be met.

A still further object of the present invention is to provide a multiple-priority non-blocking load buffer for serving a multiprocessor running real-time processes of varying deadlines by using a priority-based method to schedule memory and peripheral accesses.

5   The objects of the present invention are fulfilled by providing a data processing system comprising a first data bus for transferring data requests at a first speed, a second bus for transmitting I/O data of a second speed, and a non-blocking load buffer connected to the first and second data buses for holding the data requests and the I/O data so that a plurality of loads and stores may be performed

10   simultaneously. It is possible for the speed of the second bus to be slower than the first speed of the first bus. As a result, data may be retrieved from relatively low bandwidth I/O devices and the retrieved data may be processed and stored to low bandwidth I/O devices without idling the system unnecessarily.

In a further embodiment for a data processing system of the present

15   invention, the first data bus is connected to a processor and the second data bus is connected to a plurality of peripherals and memories. The data processing system for this embodiment reduces the effect of latency of the peripherals and memories so that the application code, which uses programmed I/O, may meet its real-time constraints. Furthermore, the code may have reduced instruction scheduling

20   constraints and a near maximum throughput may be achieved from the I/O devices of varying speeds.

Another embodiment of the present invention is fulfilled by providing a non-blocking load buffer comprising a memory array for temporarily storing data, and a control block for simultaneously performing a plurality of data loads and stores.

25   The non-blocking load buffer for this embodiment allows I/O requests from several processors to be performed at once.

Further scope of applicability of the present invention will become apparent from the detailed description given hereafter. However, it should be understood that the detailed description and specific examples, while indicating preferred

embodiments of the invention, are given by way of illustration only, since various changes and modifications within the spirit and scope of the invention will become apparent to those skilled in the art from this detailed description.

BRIEF DESCRIPTION OF THE DRAWINGS

5   The present invention will become more fully understood from the detailed description given hereinbelow and the accompanying drawings which are given by way of illustration only, and thus are not limitative of the present invention and wherein:

Figures 1(a) and 1(b) illustrate time dependency in a conventional load use

10   operation;

Figures 2(a) and 2(b) illustrate time dependency in a known Stall-On-Use operation;

Figure 2(c) illustrates time dependency in a non-blocking load buffer for an embodiment of the present invention;

15   Figure 3(a) illustrates unpredictable memory latency in a known multiprocessor system;

Figure 3(b) illustrates memory latency in a multiprocessor system having a multiple-priority version of the non-blocking load buffer;

Figure 4 illustrates a block diagram for the data processing system according

20   to one embodiment of the present invention;

Figure 5 is a schematic illustration of the non-blocking load buffer for an embodiment of the present invention;

Figure 6 illustrates an example of the contents for the entries in the memory of the data processing system for an embodiment of the present invention;

25   Figures 7(a) and 7(b) illustrate examples of the possible states for a non-blocking load through the memory of the data processing system for an embodiment of the present invention;

Figures 7(c), 7(d), 7(e), 7(f), and 7(g) illustrate the progress of addresses through queues of the data processing system in an embodiment of the present invention;

Figure 8 illustrates the circuitry used for the non-blocking load buffer in an embodiment of the present invention;

Figures 9(a) and 9(b) illustrate parallel pending queues which allow prioritization of data in the data processing system for an embodiment of the present invention;

Figure 10 illustrates a detailed block diagram of an embodiment of the non-blocking load buffer; and

Figure 11 illustrates a flow chart of the control of the pending queue in an embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The first embodiment of the present invention will be discussed with reference to Figure 4. The data processing system includes a non-blocking load buffer 10, a cache 20, one or more processors 30_0, 30_1, ..., 30_m, a processor/cache bus 40 for connecting the non-blocking load buffer 10, the cache 20 and the processors $30_{0..m}$, a plurality of peripheral devices 50_0, 50_1, ..., 50_n, and a peripheral bus 60, which includes an output bus 61 and an input bus 62, for connecting the non-blocking load buffer 10 with the plurality of peripheral devices $50_{0..n}$. Some or all of the peripheral devices $50_{0..n}$ may be memories, such as DRAM for example, which serve as a backing store for the cache 20 or as memory which the processors $30_{0..m}$ access directly through the non-blocking load buffer 10, thus bypassing the cache 20.

The processor/cache bus 40, which includes an input bus 41 and an output bus 42, transfers data from the cache 20 to registers of the processors $30_{0..m}$ as loads via the output bus 42 or alternatively, from the registers of the processors $30_{0..m}$ to the cache 20 as stores via the input bus 41. All I/O operations, which are programmed I/O, and any memory operations, even those which bypass the cache

20, are transmitted over the processor/cache bus 40 as well. For such operations the source or sink of the data is one of the peripheral devices $50_{0..n}$, such as a DRAM, for example.

The bandwidth of the peripheral bus 60 should be chosen to meet the peak input or output bandwidth which is to be expected by the peripheral devices $50_{0..n}$. The bandwidth of the non-blocking load buffer 10 should therefore be chosen to at least equal the sum of the bandwidth of the processor/cache bus 40 and the bandwidth of the peripheral bus 60.

The processors $30_{0..m}$ may be designed with dependency-checking logic so that they will only stall when data requested by a previously executed load instruction has not yet arrived and that data is required as a source operand in a subsequent instruction. Figure 2(c) illustrates the time dependency for the non-blocking load buffer 10 in an embodiment of the present invention. In Figure 2(c), load instructions are executed at operations (1), (1.1), (1.2), ... (1.n). In this example, operations (1), (1.1), (1.2), ... (1.n) correspond to:

load r1←[r2] (1);

load r5←[r2 + 24] (1.1);

load r6←[r2 + 32] (1.2);

. . . function z1 (1.n)

In this example, operation (2) executes an instruction dependent on operation (1), operation (2.1) executes an instruction dependent on operation (1.1), operation (2.2) executes an instruction dependent on operation (1.2), ... operation (2.n) executes an instruction dependent on operation (1.n). For instance, operations (2.1), (2.2), ... (2.n) correspond to:

add r3 ← r1 + r4 (2);

add r7 ← r5 + r4 (2.1);

add r8 ← r6 + r4 (2.2);

. . . function z2 (2.n).

n may be any positive integer. Operations (1.1), (1.2), ... (1.n) occur as fast as possible and the time between these operations is dependent upon the processor cycle for each operation. As shown in Figure 2(c), operation (2) waits only for a part of the latency period t_2 between operations (1) and (2), which is represented by t_i, since the part of the latency period t_2 between operations (1) and (2) is overlapped with the operations (1.1), (1.2), ... (1.n). Thereafter, operations (2.1) and (2.2) occur as fast as possible after operation (2) with the time limitation being the processor cycle. The operations (2.1) and (2.2) do not wait for the full latency period t_2 of operation (1.1) since the latency period t_2 is partially overlapped with operations (1.1), (1.2), ... (1.n) and the latency period t_2 for operation (1). For values of n sufficiently large, t_i may effectively be reduced to zero and the processor will not suffer any stalls due to memory or I/O latency. In this embodiment, there may be more than one outstanding load to slower memories or peripherals, as shown at instant t_x in Figure 2(c) for example, and requests may thereby be pipelined to provide continuous maximum throughput from the plurality of peripheral devices $50_{0..n}$ from programmed I/O of the processor.

The non-blocking load buffer 10 may accept both load and store requests from the processors $30_{0..m}$ to the peripheral devices $50_{0..n}$. The non-blocking load buffer 10 may accept requests at the maximum rate issued by a processor for a period of time that is limited by the amount of intrinsic storage in the non-blocking load buffer 10. After accepting requests, the non-blocking load buffer 10 then forwards the requests to the peripheral devices $50_{0..n}$ at a rate which the peripheral devices $50_{0..n}$ may process them. However, the non-blocking load buffer 10 does not have to wait until all of the requests are accepted.

Figure 5 illustrates internal architecture for the non-blocking load buffer in an embodiment of the present invention. The internal architecture of the non-blocking load buffer 10 includes a plurality of variable-length pending queues 114_0, 114_1, ...114_n which contain entries that represent the state of requests made by the processors $30_{0..m}$ to the peripheral devices $50_{0..n}$. The pending queues $114_{0..n}$

correspond to each of the peripheral devices $50_{0..n}$ and are used to hold the address and control information for a load or a store before being transmitted to the corresponding peripheral device. If an entry in the pending queues $114_{0..n}$ contains a store transaction, then that entry will also contain the data to be written to the

5   corresponding peripheral device. Another component of the non-blocking load buffer 10 is a variable-length return queue 116 used to buffer the data returned by any of the peripheral devices $50_{0..n}$ in response to a load until the data can be transmitted to the requesting processor. In addition, there is a free pool 112 that contains any unused request entries. Once a load request is sent from one of the

10   pending queues $114_{0..n}$ to its corresponding peripheral device, the load request is marked as "outstanding" until a return data value is sent by the peripheral and enqueued in the return queue 116. At any time, a given request entry is either outstanding, in one of the pending queues $114_{0..n}$, in the return queue 116, or is not in use and is therefore located in the free pool 112.

15   Figure 6 illustrates an example of a memory structure 200 for storing the information present in each entry of the non-blocking load buffer 10. E entries may be stored in the non-blocking load buffer 10 and each entry holds one of a plurality of requests 201_1, 201_2, ...201_E which can either be a load or a store to any of the peripheral devices $50_{0..n}$. Each request 201 includes a control information 202, an

20   address 204 and data 206. However, for a load operation, the data 206 is empty until data returns from the peripheral devices $50_{0..n}$. The control information 202 describes parameters such as whether the transaction is a load or a store, the size of data transferred, and a coherency flag that indicates whether all previous transactions must have been satisfied before the new transaction is permitted to be

25   forwarded to the addressed peripheral device. The coherency flag enforces sequential consistency, which is occasionally a requirement for some I/O or memory operations. The data stored in the entries 201 need not be transferred among the queues of the non-blocking load buffer 10. Rather pointers to individual entries

within the memory structure 200 may be exchanged between the pending queues $114_{0..n}$, the return queue 116 and the free pool 112 for example.

Figures 7(a) and 7(b) illustrate examples of the operation of the non-blocking load buffer 10 which will be input to a peripheral device such as a RAM. Initially,

5   all entries of the non-blocking load buffer 10 are unused and free. With reference to Figure 7(a), when one of the processors $30_{0..m}$ issues a non-blocking load, a free entry is chosen and the address of the load is stored in this entry. Additionally, space for the returning data is allocated. This entry now enters the pending queue, one of the queues $114_{0..n}$, corresponding to the addressed peripheral device. When

10   the peripheral device is ready, the pending entry is delivered over the peripheral bus 60 and becomes an outstanding entry. When the outstanding entry returns from the peripheral device via the peripheral bus 60, the accompanying requested load data is written to the part of the entry that had been allocated for the returning data and the entry is placed in the return queue 116. The data is buffered until the one

15   processor is ready to receive the data over the processor/cache bus 40 (*i.e.*, until the one processor is not using any required cache or register file write ports). Finally, the data is read out and returned to the one processor over the processor/cache bus 40 and the entry again becomes free. In addition, a peripheral device may accept multiple requests before returning a response.

20   ′ Figure 7(b) illustrates the progress of an entry during a store operation. The store operation is similar to the load operation but differs because data does not need to be returned to the processors $30_{0..m}$ and this return step is, therefore, skipped. Once a store is accepted by the non-blocking load buffer 10, the store is treated by the issuing processor as if the data has already been transferred to the target

25   peripheral device, one of the peripheral devices $50_{0..n}$.

The non-blocking load buffer 10 performs similarly to a rate-matching FIFO but differs in several ways. For instance, one difference is that although memory transactions forwarded by the non-blocking load buffer 10 are issued to any particular peripheral device $50_{0..n}$ in the same order that they were requested by any

of the processors $30_{0..m}$, the transactions are not necessarily FIFO among different peripheral devices. This is similar to having a separate rate-matching FIFO for outputing to each of the peripheral devices $50_{0..n}$ except that the storage memory for all the FIFOs is shared and is thereby more efficient. Also, the peripheral devices

5      $50_{0..n}$ are not required to return the results of a load request to the non-blocking load buffer 10 in the same order that the requests were transmitted.

The non-blocking load buffer 10 is capable of keeping all of the peripheral devices $50_{0..n}$ busy because an independent pending queue exists for each of the peripheral devices. A separate flow control signal from each of the peripheral

10   devices indicates that this corresponding peripheral device is ready. Accordingly, a request for a fast peripheral device does not wait behind a request for a slow peripheral device.

Figures 7(c), 7(d), 7(e), 7(f) and 7(g) illustrate examples of a non-blocking load progressing through the queues. After the queues are reset, entries 1, 2, 3 and

15   4 are in the free pool as shown in Figures 7(c). A non-blocking load is pending after the entry 1 is written into one of the pending queues as shown in Figure 7(d). Next, Figure 7(e) illustrates that entry 1 becomes outstanding after the pending load is accepted by its corresponding peripheral device. Subsequently, the load is returned from the peripheral device and entry 1 is now in the returned queue as

20   illustrated in Figure 7(f), and awaits transmission to the requesting processor (or to the cache if the access was part of a cache fill request). Finally, in Figure 7(g), entry 1 is released to the free pool and, once again, all entries are free.

Figure 8 illustrates the circuitry used for the non-blocking load buffer in an embodiment of the present invention which implements variable depth FIFOs. This

25   circuitry manipulates pointers (i.e. RAM addresses for example) to entries in a RAM 130 instead of storing the addresses of only the head and tail of a contiguous list of entries in the RAM 130. Furthermore, the circuitry does not require the pointers to be sequential. Therefore, records need not be contiguous and the RAM 130 can contain several interleaved data structures allowing more flexible allocation

and de-allocation of entries within a FIFO so that it is possible to use the same storage RAM 130 for multiple FIFOs. The entire circuit in Figure 8 represents a large FIFO of an unspecified data width which is controlled by a smaller FIFO represented by block 170 having pointer widths smaller than the records. In Figure

5   8, a first control circuit 110, a second control circuit 120, the RAM 130 having separate read and write ports, a plurality of shift registers 150_1, 150_2, 150_3, ...150_n corresponding to each entry in the RAM 130 and a multiplexer 160 are illustrated. The control circuits 110 and 120 provide pointers to the entries of the RAM 130 rather than the entries themselves being queued. Block 170 is essentially a FIFO

10   of pointers. A pointer may be enqueued in the FIFO of pointers in block 170 by asserting the "push" control line active and placing the pointer (SRAM 130 address) on the "push-addr" input lines to block 170. Similarly, a pointer value may be dequeued by asserting the "pop" control line on block 170 with the pointer value appearing on the "pop-addr" bus. Block 170 has a "full" output that when active

15   indicates that no more pointers may be enqueued because all the registers $150_{1..p}$ are occupied and the queue is at its maximum depth, p. Block 170 also has an "empty" output that indicates when no pointers remain enqueued.

In a preferred embodiment, the RAM 130 is divided into two arrays, an address control array AC and a data array D. The address control array AC stores

20   the control information 202 and the address 204 for each entry and the data array D stores the data 206 for each entry. Dividing the non-blocking load buffer into these two arrays AC and D realizes a more efficient chip layout than using a single array for both purposes.

Figure 10 illustrates the entire structure for a non-blocking load buffer in an

25   embodiment of the present invention. Each of the pending queues $114_{0..n}$ and the return queue 116 may be composed of copies of block 170 illustrated in Figure 8 for example. The queue entries are stored in the AC and D arrays 501 and 502, with pointers to the array locations in the flip-flops that compose the pending queues $114_{0..n}$, the return queue 116 and the free pool 112. For the embodiment illustrated

in Figure 10, the write ports to the AC and D arrays are time-division multiplexed using multiplexers 301 and 302 to allow multiple concurrent accesses from the processors $30_{0..m}$, which issue load and store requests over bus 41, and the peripheral devices $50_{0..n}$, which return requested load data over bus 62. Similarly,

5   the read ports are time-division multiplexed by multiplexer 303 between returning load data going back to the processors $30_{0..m}$ via bus 42 and issuing of load and store requests to the peripherals $50_{0..n}$ via bus 61. A processor request control unit 510 accepts the load and store requests from the processors $30_{0..m}$, a return queue control unit 520 controls the data returned by the peripheral devices $50_{0..n}$ to the return

10   queue 116 and a pending queue control unit 530 controls the selection of the pending queue $114_{0..n}$.

When either a non-blocking load buffer or a conventional single FIFO load buffering operation is used, the uncertainty and memory latency is further compounded because individual processors may transmit more than one memory

15   request at a time as illustrated in Figure 3(a) for example. In Figure 3(a), processor A issues three non-blocking loads to a memory device after a first non-blocking load is issued by processor B but before a second non-blocking load is issued by processor B to the same memory device. The first memory transaction issued by processor B, labeled B1, will be executed immediately. However, the latency of

20   the second transaction, labeled B2, by processor B, t_{B2}, will be considerably longer than the latency of the first transaction, t_{B1}, since the second transaction B2 will not be handled by the memory system until all of the requests A1, A2, and A3 by processor A have been processed.

One consideration for overcoming all of these unpredictable memory latency

25   effects is to have the programmer carefully design the times at which each real-time process attempts to access the memory which effectively has software serialize access to the memory banks instead of hardware. For instance, two processors may alternate memory accesses at regular intervals to match the throughput capabilities of the memory system. However, such programming places a heavy burden on both

the program and the programmer because the programmer must be aware of the detailed temporal behavior of the memory access patterns by the program. The behavior of the programs is usually very complex and may change based on its input data, which is not always known. Therefore, it is impractical for the programmer

5   to know the detailed temporal behavior of the memory access patterns by the program.

Also, because the combinations of processors that are making the requests are even more difficult to predict and the combinations of programs running on the different processors might not always be the same on each occasion or even known,

10   the detailed temporal behavior of the memory access patterns by the program may not be known. For example, two processes that run two different programs on two different processors may have been written by different programmers and the source code may not even be available for either or both programs. As a result, this programming approach for overcoming the unpredictable memory latency defeats

15   the purpose of using a non-blocking load buffer, which was designed to simplify the burden on the programmer and the compiler by relaxing the scheduling constraints under which memory operations can be issued while still permitting efficient utilization of the processor.

Figure 9(a) illustrates another embodiment of the present invention for a

20   multiple-priority version of the non-blocking load buffer. In this embodiment, the non-blocking load buffer is a variation of the basic non-blocking load buffer where multiple pending sub-queues 214_0, 214_1, ...214_p exist as a component of each pending queue $114_{0..n}$. The outputs of the sub-queues $214_{0..p}$ are then input to the sub-multiplexers 218 associated with one of the peripheral devices. Each output of

25   the sub-multiplexers 218 are then input to the main multiplexer 118. Each of these pending sub-queues is assigned a unique priority level. In the simplest implementation of this multiple-priority version of the non-blocking load buffer, illustrated in Figure 9(b), there are two pending sub-queues 214_0 and 214_1 for a peripheral device with sub-queue 214_0 being assigned a high priority and sub-queue

214_1 assigned a low priority. The multiple priority non-blocking load buffer issues memory or peripheral transactions in a highest-priority-first manner. In other words, no transaction will be issued from a pending sub-queue for a given peripheral unless all of the higher priority sub-queues for that same peripheral are

5   empty. For each peripheral device, requests are issued from the pending sub-queue in a FIFO manner for memory transactions of the same priority.

The computational processes associated with the memory transactions that take place at each of these relative priority levels execute on the processors $30_{0..m}$. The priority levels are assigned by the processors based on the scheduling

10   parameters of all the processes in the system. The priority level of the process is designed to reflect the relative importance of the memory accesses with respect to allowing all of the processes to meet their deadlines if possible. The priority levels may also be applicable in a uni-processor environment. For example, interrupt handlers may be provided to achieve low latency by using higher priority memory

15   transactions than the transaction being interrupted. The priority level may be identified by adding a priority tag to the memory transaction. This priority tag is used to channel the memory transaction into the pending sub-queue with the matching priority level, thus the selection of the appropriate destination pending sub-queue for a non-blocking memory transaction is a function of both the address and

20   the priority level of the access. The priority tag may be stored with the other control information 202 in the queue entry 201 corresponding to a given non-blocking memory access.

Figure 11 illustrates a flow chart for the functions performed by the pending queue control unit in an embodiment of the present invention for the multiple-

25   priority version of the non-blocking load buffer. At step S10, the counter i for a pending queue is initialized to zero. At step S20, the counter is compared to the number n of pending queues 114. If the counter is not equal to the total number of pending queues, the counter is incremented at step S30 and a determination is made at step S40 of whether the peripheral corresponding to the counter is ready. If the

peripheral device is determined not to be ready at step S40, the process returns to step S20. However, if the peripheral corresponding to the counter is ready at step S40, a pending queue is selected at step S50 with the highest priority that also contains at least one memory transaction. At step S60, memory transactions are

5   issued in a FIFO order from the selected sub-queue and the process returns to step S20. If the counter is equal to the number of pending queues at S20, the process has been completed for all the pending queues and step S10 is returned to where the counter is initialized.

Figure 3(b) illustrates memory latency for the multiple-priority non-blocking

10   load buffer. In this example, processor B's memory transactions are assigned a higher priority than processor A's memory transactions. Therefore, transaction B2 is delivered to the memory before transactions A2 and A3 even though the request to begin transaction B2 arrived at the non-blocking load buffer after requests A2 and A3. As a result, the latency for transaction B2, t_{B2} is less in Figure 3(b) than t_{B2} in

15   Figure 3(a), which illustrates a non-blocking load buffer that does not offer the benefit of multiple-priority scheduling. Using the multiple-priority version of the non-blocking load buffer, Processor B spends less time stalled waiting for transaction B2 to complete as illustrated by the comparison in Figures 3(a) and 3(b).

These priority levels may be heuristic in nature. For example, if using

20   earliest deadline first (EDF) scheduling, a process should not be assigned a lower priority than the priority of any process which has a more distant deadline. In general, this priority level is not necessarily fixed for each process and the priority level may vary over time as the demands of the real-time process or of other processes change. As another example of a priority selection mechanism, if the

25   load-use distance is known for a load instruction (as computed by a compiler), it can be used to set a priority for each individual memory access instruction. (Higher load-use distances result in lower priorities.) In general, non real-time processes (the processes having no deadlines) are typically given the lowest priority.

Process priority may also be used to arbitrate for the limited resources within the non-blocking load buffer itself. For example, in one embodiment of the non-blocking load buffer, limited data storage memory is shared among all peripheral devices and processors. When the total number of slots among all of the non-

5   blocking load buffer queues exceeds the number of non-blocking memory entries, it is possible for a process at a low priority to prevent a higher priority process from completely utilizing one or more of its pending sub-queues by using up these entries. The use of priority in allocating non-blocking memory entries can be used to eliminate this effect. For example, the maximum number of outstanding non-

10   blocking memory transactions may be specified for each of the available priorities.

Once an appropriate priority has been determined for a process, the priority of its memory transactions might be specified to the non-blocking load buffer by employing any of several techniques. For example, the priority of any given memory transaction might be determined by an operand to the instruction that

15   performs the memory access or the priority can be associated with certain bit combinations in either the virtual address, the physical address of the memory or the physical address of the peripheral device accessed. Alternatively, the processors might be designed with programmable registers that set the priority of all memory accesses for each processor. Yet another possible technique is to store the memory

20   access priority in the page table of a virtual memory system and thus make the memory access priority a characteristic of specific memory pages. When these various techniques are combined with conventional memory management and processor design techniques, memory priorities can be treated as privileged resources. As a result, the operating system reserves the highest priority levels for

25   its own real-time tasks and therefore the user level programs are extremely limited in their ability to disrupt important system real-time tasks.

To maintain memory coherency in a multiple-priority non-blocking load buffer, which is not always necessary, no load or store may be issued to any of the peripheral devices while a load or store to the same address is outstanding or while

an earlier load or store to the same address is pending. The memory coherency may be maintained by time stamps combined with address comparison. However, memory coherency may also be maintained more simply by ensuring that requests of different priorities are sent to non-overlapping address segments within the same

5   peripheral device.

In the normal operation of programmed I/O activity, the processors $30_{0..m}$ do not need to exactly schedule loads to achieve maximum throughput from the peripheral devices but can instead burst out requests to the limits of the non-blocking load storage and, if properly programmed, perform useful work while

10   waiting for data to return. The non-blocking load buffer allows application code to use programmed I/O (memory mapped I/O) for achieving near maximum throughput from the I/O devices of varying speeds which reduces the effective latency of the I/O peripheral devices and relaxes the scheduling constraints on the programmed I/O. The non-blocking load buffer functions to rate-match the requests from the

15   processor to the peripheral devices and back and to act upon the priority of requests from the processor to allow high priority requests to go ahead of low priority requests already buffered.

The non-blocking load buffer uses queues of pointers to centralize storage to increase storage density, parallel queues to implement requests of different

20   priority and memory segment descriptors to determine priority. Accordingly, I/O requests to a fast device need not wait behind requests to a slow device and requests from several processes of a signal processor running multiple processes are buffered so that the processor is not unnecessarily idled and the time to complete tasks is reduced. Because the multiple priority non-blocking load buffer has multiple

25   pending sub-queues for each of the peripherals, a processor used in combination with this multiple priority non-blocking load buffer is able to run real-time processes of varying deadlines by use of non-FIFO scheduling of memory and peripheral accesses. Also, the multiple priority non-blocking load buffer simplifies the burden on the programmer and the compiler by relaxing the scheduling constraints under

which memory operations can be issued while still permitting efficient utilization of the processor.

The invention being thus described, it will be obvious that the same may be varied in many ways. Such variations are not to be regarded as a departure from the spirit and scope of the invention, and all such modifications as would be obvious to one skilled in the art are intended to be included within the scope of the following claims.